ChatGPT

Python DSAI Lab Exam Cheatsheet

Object-Oriented Programming (OOP)

Core Concepts: Python uses *classes* to model real-world entities, bundling data (attributes) and functions (methods). A class defines a blueprint, and an *object* (instance) is created from this blueprint. Key syntax includes the `class` keyword, an `__init__` constructor for initializing instances, and `self` to refer to the instance inside class methods. Classes support principles like **encapsulation** (grouping data with methods), **inheritance** (deriving new classes from existing ones), and **polymorphism** (methods behaving differently based on context). Use OOP to structure complex data (e.g., students in departments in an institute) in a logical, modular way.

Defining a Class: Inside `__init__`, assign attributes to `self`. Define methods to operate on those attributes. Special methods (dunder methods) like `__str__` / `__repr__` can provide readable string representations, and comparison methods like `__lt__`, `__gt__`, etc., enable object comparisons. For example, implementing `__gt__` allows using `>` between objects (e.g., compare students by average score). Python also provides the `dataclasses` module to auto-generate common methods like `__init__` and `__repr__` for classes intended to primarily store data [1].

Common Patterns & Tips:

- **Composition:** Build complex structures by composing classes. *Example:* An `Institute` has multiple `Department` objects, each containing a list of `Student` objects (Student→Department→Institute structure).
- **Special Methods:** Implement `__str__` or `__repr__` in classes to make debugging/printing easier. Implement comparison methods (`__eq__`, `__lt__`, `__gt__`) if you need to sort or compare objects by a certain attribute.
- **Class vs Instance Attributes:** Use class attributes for properties shared by all instances; use instance attributes (`self.attribute`) for per-object data. Avoid mutable class attributes that could be unintentionally shared across instances.
- **Lesser-known features:** Use `@staticmethod` for utility functions within a class that don't use `self`, and `@classmethod` for alternative constructors or methods needing class-level access (via `cls`). The `@dataclass` decorator (in Python 3.7+) can reduce boilerplate for simple data-holder classes [1].
- **Common Pitfalls:** Forgetting to include `self` as the first parameter in instance methods (leading to errors), or accidentally using mutable default arguments in `__init__` (which can be shared across instances).

```python
import numpy as np

# Example classes for Student and Department
class Student:
    def __init__(self, name, roll_number, scores):
        self.name = name
        self.roll_number = roll_number
        self.scores = np.array(scores, dtype=float)  # store scores in NumPy
```

```
array
    def average(self):
        return np.mean(self.scores)                 # average score
    def highest_score(self):
        return np.max(self.scores)
    def lowest_score(self):
        return np.min(self.scores)
    def __gt__(self, other):
        # Compare students by average score for sorting/ranking
        return self.average() > other.average()
    def __repr__(self):
        return f"Student({self.name}, avg={self.average():.1f})"

class Department:
    def __init__(self, name):
        self.name = name
        self.students = []
    def add_student(self, student):
        self.students.append(student)
    def topper(self):
        # Return student with highest average (uses Student.__gt__)
        return max(self.students)

# Create students and a department
s1 = Student("Alice", 1, [90, 95, 85, 80])
s2 = Student("Bob",   2, [70, 88, 90, 85])
print(s1.name, "avg=", s1.average())        # Alice avg= 87.5
print(s2.name, "avg=", s2.average())        # Bob avg= 83.25
print(s1 > s2)                              # True  (uses __gt__ to compare
averages)
dept = Department("CS")
dept.add_student(s1); dept.add_student(s2)
topper = dept.topper()
print("Dept Topper:", topper)               # Dept Topper: Student(Alice,
avg=87.5)
```

Above: We define a `Student` class with methods to compute average and other stats using NumPy, and override `__gt__` to compare students by average score. The `Department` class contains students and can determine the top student. This illustrates composition (Department has many Students). In practice, you might also implement an `Institute` class containing departments, and methods like `institute_average()` or searching a student by roll number across all departments.

If a class is mainly for storing data without complex behavior, consider using a dataclass for brevity [2]:

```
from dataclasses import dataclass

@dataclass
class Course:
    code: str
    credits: int = 0  # default value
```

```
    instructor: str = ""
# This automatically provides an __init__ and a nice __repr__.
c = Course("DSAI101", 4, "Dr. Smith")
print(c)  # Course(code='DSAI101', credits=4, instructor='Dr. Smith')
```

NumPy

Core Concepts: NumPy provides the `ndarray` object, a high-performance n-dimensional array for numerical computing. Using NumPy arrays allows **vectorized** operations, meaning you can apply operations on whole arrays without Python loops (which is much faster). Arrays have a shape (tuple of dimensions) and support element-wise arithmetic, slicing, and a variety of mathematical functions.

Creating Arrays: You can create arrays from Python lists or tuples using `np.array()`. There are also convenience functions for ranges: `np.arange(start, stop, step)` (like Python's range but returns array), and `np.linspace(start, stop, num)` for evenly spaced numbers. For example, `np.zeros((3,4))` creates a 3x4 array of zeros; similarly `np.ones` and `np.eye` (identity matrix). Random arrays can be generated via `np.random` (e.g., `np.random.rand(3,4)` for uniform [0,1) randoms, or `np.random.normal(mean, sd, size)` for Gaussians).

Array Operations and Indexing: Arithmetic on arrays is element-wise. Expressions like `array * 2` or `array1 + array2` apply to each element (arrays will **broadcast** to compatible shapes automatically). Use standard Python slicing syntax to access subsets: `a[0:3]` gives first 3 elements (1D), and for multi-dimensional arrays use `a[row_index, col_index]`. You can slice each dimension: e.g. `matrix[:, 0]` selects the first column of a 2D array, `matrix[1, :]` the second row. Boolean indexing is very powerful: you can use a condition to filter, e.g. `a[a > 0]` returns all positive elements of `a`. This creates a boolean mask array that selects only True positions.

Common Functions: NumPy has optimized functions for aggregations: `np.sum`, `np.mean`, `np.min`, `np.max`, `np.std`, etc., which can operate on the whole array or along a given axis (e.g., `np.sum(matrix, axis=0)` sums each column). Use `np.argmax` or `np.argmin` to get the index of the max or min value (for multi-dim arrays, specify axis or it will return a flattened index). Sorting can be done with `np.sort` (returns a sorted copy) or `np.argsort` (returns indices that would sort the array). You can reshape arrays with `arr.reshape(new_shape)` (the total number of elements must match) or flatten with `arr.ravel()` / `arr.flatten()`. To combine arrays, use `np.concatenate` or stack functions (`np.vstack`, `np.hstack`, `np.column_stack`, etc.).

Lesser-Known Tips:

- **Broadcasting:** NumPy can automatically *broadcast* smaller arrays to align with bigger ones in arithmetic. For instance, adding a 1D array of length 3 to a 3x3 matrix will add that 1D array to each row of the matrix without explicit looping.
- **Vectorized Computations:** Replace Python loops with NumPy operations. For example, computing BMI for a list of people can be done with array math instead of looping over each person.
- **Boolean Masks:** Use conditions directly on arrays to create masks (e.g., `mask = arr < 0`), then use `arr[mask]` to select elements meeting the condition. Combine conditions with bitwise operators (`&`, `|`) and parentheses, e.g. `arr[(arr > 0) & (arr < 10)]`.

- **np.where:** This is useful for conditional selection. `np.where(condition, x, y)` will form an array choosing elements from `x` or `y` based on the boolean `condition` (like a vectorized ternary operator).
- **Random Seed:** For reproducibility in random number generation, call `np.random.seed(some_number)` before generating random arrays.

Common Pitfalls: NumPy arrays are homogeneous (all elements are the same type), so mixing types can lead to upcasting (e.g., mixing ints and floats yields floats). Indexing a multi-dim array incorrectly (e.g., using a single index on a 2D array returns a 1D row). Also, modifying a slice of an array modifies the original data (since slices are views, not copies) – use `arr.copy()` if you need a separate copy.

```
import numpy as np

# Example 1: BMI calculation using vectorized operations
heights_in = np.array([65, 70, 75])            # heights in inches
weights_lb = np.array([150, 180, 210])         # weights in pounds
heights_m = heights_in * 0.0254                # convert inches to meters
weights_kg = weights_lb * 0.453592             # convert lb to kg
bmi = weights_kg / (heights_m ** 2)            # BMI = weight(kg) /
height(m)^2
print(bmi)            # -> [24.96 25.83 26.25]  (BMI values for each person)
# Create a boolean mask for BMI < 25 (e.g., to find underweight individuals)
mask = bmi < 25
print(mask)           # -> [ True False False]
print(bmi[mask])      # -> [24.96]  (BMIs that are below 25)

# Example 2: 2D array creation and indexing
matrix = np.array([[1,2,3],
                   [4,5,6],
                   [7,8,9]])
print(matrix.shape)   # -> (3, 3)  (rows, cols)
print(matrix[1, :])   # -> [4 5 6]   second row
print(matrix[:, -1])  # -> [3 6 9]   last column

# Example 3: Aggregations and axis parameter
col_sums = np.sum(matrix, axis=0)
row_means = np.mean(matrix, axis=1)
print(col_sums)       # -> [12 15 18]  (sum of each column)
print(row_means)      # -> [2. 5. 8.]  (mean of each row)

# Example 4: Using random and argsort/argmax
np.random.seed(0)                          # set seed for reproducibility
rand_arr = np.random.randint(1, 10, size=(2, 3))
# 2x3 array of random ints 1-9
print(rand_arr)
# e.g., rand_arr -> [[6 1 4]
#                    [4 8 4]]
print(np.max(rand_arr))           # -> 8        (maximum value in the whole
array)
print(np.argmax(rand_arr))        # -> 4        (index of max in flattened
array)
```

```
print(np.argmax(rand_arr, axis=1))# -> [0 1]    (indices of max in each row)
# Get indices of top-2 values in each row using argsort
top2_indices = np.argsort(rand_arr, axis=1)[:, -2:]
print(top2_indices)    # -> [[2 0]
                       #     [2 1]]  (columns of the two largest values per
row)
```

Above: We demonstrate creating arrays and performing operations without loops. In Example 1, height and weight arrays are converted to metric units and BMI is computed in one go; we then filter out BMIs below a threshold using a boolean mask. Example 2 shows 2D array indexing and shape. Example 3 computes sums and means across rows/columns by specifying the `axis`. Example 4 generates a random 2x3 array of integers (seeded for consistent results), finds maxima, and uses `np.argsort` to identify indices of the top 2 values per row (a pattern useful for tasks like finding top-performing weeks/products, etc.).

Pandas

Core Concepts: Pandas provides two primary data structures: **DataFrame** (tabular data with named columns and row indices) and **Series** (a 1D labeled array). It's widely used for data cleaning, manipulation, and analysis. A DataFrame is like a spreadsheet or SQL table, enabling easy reading/writing of data and powerful operations like grouping, joining, and time-series handling.

Creating and Loading Data: You can create a DataFrame from a dict of lists or from NumPy arrays. More often, you'll load data from files using `pd.read_csv("file.csv")` (or `read_excel`, etc.). For example, `df = pd.read_csv("data.csv")`. Once loaded, inspect the data with `df.head()` (first 5 rows), `df.tail()`, `df.shape` (rows, columns), `df.columns`, and `df.info()` to see data types and non-null counts. `df.describe()` gives summary statistics for numeric columns.

Indexing and Slicing: There are multiple ways to select data: - By column: `df['ColumnName']` returns a Series. For multiple columns, use a list: `df[['Col1','Col2']]` returns a DataFrame with those columns. - By row label or integer location: use `df.loc[row_label, col_label]` for label-based indexing, and `df.iloc[row_index, col_index]` for position-based. For example, `df.loc[0, 'Age']` gets the 'Age' of the first row (index 0) and `df.iloc[0:3, :2]` gets the first 3 rows and first 2 columns. - Boolean filtering: e.g., `df[df['Age'] > 30]` returns only rows where Age > 30. Combine conditions with `&` (and), `|` (or) and wrap each condition in parentheses: `df[(df['Age']>30) & (df['Survived']==1)]`. You can also use `df.query('Age > 30 and Survived == 1')` as a shortcut. - Setting values: Assign to selections using `loc`. For example, `df.loc[df['Age'] < 0, 'Age'] = None` (this would mark negative ages as missing). Be cautious with chained indexing like `df[df['Age']>30]['Fare'] = 0` – it may cause SettingWithCopy warnings. Prefer `loc` to both filter and assign in one go.

Data Cleaning: Pandas has tools to deal with missing or anomalous data: - Detect missing with `df.isnull()` (or `df.notnull()`). Drop missing values via `df.dropna()`, or fill them with a value using `df.fillna(value)` or with summary statistics like median/mean: `df['Age'].fillna(df['Age'].median(), inplace=True)`. - You can drop entire columns or rows. For columns: `df.drop(columns=['Cabin', 'Ticket'])` removes those columns. For rows: `df.drop(index=[0,1])` drops rows with those index labels. - Outliers can be detected by conditions (e.g., `df[df['Fare'] > 300]`) or using statistical methods (z-score or IQR). For example, find outliers by Z-score: `df[(df['Fare'] - df['Fare'].mean()).abs() > 3*df['Fare'].std()]`. -

Create new columns as needed: `df['Fare_per_Age'] = df['Fare'] / df['Age']` (will be NaN if Age is NaN). You can use `df.assign(NewCol=...)` as well, which is handy in method chaining.

Grouping and Aggregation: Use `df.groupby()` to group rows by one or more keys and then aggregate. For example, `df.groupby('Survived')['Age'].mean()` gives average age by survival status. You can aggregate multiple columns: `df.groupby('Survived').agg({'Age':'mean', 'Fare':'median'})`. Common aggregation functions: `mean()`, `sum()`, `count()`, `min()`, `max()`, `median()`, `std()` etc. You can also apply custom functions. The result of groupby is an object you can iterate over or directly convert to a new DataFrame.

Method Chaining and Pipeline: Pandas operations can be chained in one expression for clarity. For instance, you can do:

```
result = (df.drop(columns='Cabin')
            .fillna({'Age': df['Age'].median()})
            .groupby('Survived')[['Age','Fare']].mean())
```

This reads like a pipeline of transformations. Another approach is using the DataFrame `.pipe()` method to send the DataFrame through a function. For example, define functions `handle_missing(df)`, `detect_outliers(df)`, `summarize(df)` and then do: `df.pipe(handle_missing).pipe(detect_outliers).pipe(summarize)`. This is useful for sequential data transformations [3] and keeps code readable.

Lesser-Known but Useful: - `.apply()` **and** `.applymap()`**:** Apply a function to each row or column (`df.apply(func, axis=1)` for rows) or element-wise to a DataFrame (`applymap`). - `.astype()`**:** Convert data types, e.g. `df['Age'] = df['Age'].astype('int')` after filling missing. - `.merge()`**:** Join DataFrames (like SQL JOIN) on a key column: `pd.merge(df1, df2, on='id', how='inner')`. - **Reshaping:** Use `pd.pivot_table` or `df.pivot` to reshape data (convert from long to wide format, etc.), and `pd.melt` to go from wide to long. - **Datetime Handling:** Pandas has powerful datetime features (if a column is dates, convert with `pd.to_datetime`, then you can access properties like `.dt.year`, and resample time series with `df.resample('M').mean()` if you set a DateTime index).

Common Pitfalls: Forgetting that most Pandas methods return a new DataFrame/Series (and thus needing to assign the result or use `inplace=True` for in-place operations). Chained indexing (as mentioned) can lead to bugs or warnings. Also, mixing up `loc` vs `iloc` – remember `loc` is label-based (and inclusive of end label in slicing), `iloc` is integer position-based (exclusive of end index in slicing, like regular Python ranges).

```
import pandas as pd
import numpy as np

# Create a small DataFrame (e.g., resembling a snippet of Titanic data)
df = pd.DataFrame({
    'Age':      [22, 38, 26, np.nan, 40],
    'Fare':     [7.25, 71.83, 7.925, 8.05, 15.50],
    'Survived': [0, 1, 1, 0, 1],
    'Cabin':    ['C85', None, None, 'C123', 'E33']
```

```
})
print(df.head())          # View the first 5 rows

# Basic info
print(df.shape)           # -> (5, 4)
print(df.columns)         # -> ['Age', 'Fare', 'Survived', 'Cabin']
# df.info()  # (Would display data types and counts; omitted in this output)

# Fill missing Age with median, drop the Cabin column
df['Age'].fillna(df['Age'].median(), inplace=True)
df.drop(columns=['Cabin'], inplace=True)

# Create a new column (Fare per year of age, just as an example)
df['Fare_per_Year'] = (df['Fare'] / df['Age']).round(3)
print(df)  # Data after cleaning and new column addition

# Group by survival status and compute average Age and Fare
grouped = df.groupby('Survived')[['Age', 'Fare']].mean().round(2)
print(grouped)
```

Output after cleaning and new column:

```
    Age    Fare  Survived  Fare_per_Year
0  22.0   7.25         0          0.330
1  38.0  71.83         1          1.890
2  26.0   7.925        1          0.305
3  32.0   8.05         0          0.252
4  40.0  15.50         1          0.388
```

The grouping result (`grouped`) might look like:

```
            Age   Fare
Survived
0         27.00   7.65
1         34.67  31.75
```

This indicates, for example, that those who did not survive (`Survived=0`) had an average Age ~27 and average Fare 7.65, whereas survivors had mean Age ~34.67 and Fare ~31.75 (note: small dataset example).

Explanation: We loaded a DataFrame and saw basic structure. We then filled missing Age with the median (here median age became 32) and dropped the Cabin column (due to too many missing values). Next, we added a computed column (`Fare_per_Year`) as Fare divided by Age (rounded to 3 decimals for clarity). Finally, using `groupby`, we calculated average Age and Fare by survival status. This pattern of **load → clean → derive new columns → analyze** is common in data analysis tasks. Remember, you can chain operations or use intermediate variables as needed for clarity.

Matplotlib

Core Concepts: Matplotlib is the fundamental plotting library in Python for creating static, interactive, and animated visualizations. Its syntax is heavily inspired by MATLAB. The most common interface is `matplotlib.pyplot` (imported as `plt`), which provides state-based functions to create figures and plots. Matplotlib operates on a Figure (the entire image or canvas) which contains one or more Axes (subplots). You can use the stateful **pyplot** API (`plt.plot`, `plt.show`, etc.) or the object-oriented API (`fig, ax = plt.subplots(); ax.plot(...)`). For quick plotting, pyplot is simple; for more complex layouts, the object-oriented approach is preferred.

Basic Plot Types: - **Line Plot:** Use `plt.plot(x, y)` to plot y vs x as a line (defaults to connecting points). Great for time series or continuous data trends. You can customize line style (`linestyle` or shorthand like '--' for dashed), color, and markers for data points. - **Scatter Plot:** Use `plt.scatter(x, y)` to plot unconnected points, useful for showing relationship between two variables. - **Bar Chart:** `plt.bar(x, height)` for vertical bars (or `plt.barh` for horizontal). Commonly used for categorical data comparison. - **Histogram:** `plt.hist(data, bins=...)` to show distribution of a single variable. - **Pie Chart:** `plt.pie(values, labels=...)` to show parts of a whole (with caution for too many categories). - **Box Plot:** For distribution summary (min, Q1, median, Q3, max, outliers), you can use Matplotlib's `plt.boxplot(data)` or use Seaborn for a higher-level interface. - **Other:** Matplotlib can also do area charts, stack plots, stem plots, etc., but line, scatter, bar, pie, histogram cover many needs.

Customization: Add titles and axis labels with `plt.title("Title")`, `plt.xlabel("X Label")`, `plt.ylabel("Y Label")`. Include a legend if multiple data series: either by calling `plt.legend()` after plotting with label arguments, or by Axes method. Adjust ticks (rotation for dates or categories) via `plt.xticks(rotation=45)` for example. Add reference lines using `plt.axhline(y=value)` or `plt.axvline(x=value)` for horizontal/vertical lines (e.g., threshold lines). Annotate points with `plt.text(x, y, "Label")` to place custom text. Control figure size by `plt.figure(figsize=(width,height))` or when using subplots. Use `plt.xlim, plt.ylim` to set axis ranges. Matplotlib also supports log scales (`plt.yscale('log')`) and twin axes for dual scales.

Subplots: To create multiple plots in one figure, use `plt.subplot` or `plt.subplots`. For example, `fig, axs = plt.subplots(2, 2)` creates a 2x2 grid of subplots, and `axs[0,1]` refers to the Axes in first row, second column. You can then treat each `ax` like `plt` (e.g., `axs[0,1].plot(...)`, `axs[0,1].set_title("...")`). Use `plt.tight_layout()` to adjust spacing. You can give the entire figure a super-title with `fig.suptitle("Overall Title")`. When done, `plt.show()` will render all subplots together.

Saving Figures: Use `plt.savefig("filename.png")` to save the current figure to file (call before `plt.show()` to avoid potential formatting differences). You can specify DPI and other parameters for higher resolution if needed.

Tips and Lesser-Known Tricks: - **Styles:** Matplotlib has predefined styles (e.g., `plt.style.use('ggplot')` or `'seaborn'`) to instantly improve aesthetics. - **Figures and Axes:** If making many plots in code, explicitly create and close figures (`plt.figure()` or use `with plt.style.context(...)`) to avoid overlapping or memory issues. - **Annotation:** The `plt.annotate` function can add text with arrows pointing to data points (useful to highlight specific points). - **Interactive Plotting:** In Jupyter, use `%matplotlib inline` (for static images) or

`%matplotlib notebook` (for interactive zoom) if needed. - **Limit Display Range:** For clarity, you might restrict the axis range with `plt.xlim(min, max)` or use `plt.ylim` for y-axis. - **Twin Axis:** `ax.twinx()` or `ax.twiny()` can create a second y-axis or x-axis in the same plot (useful when plotting variables with different scales on the same chart).

Common Pitfalls: Forgetting to call `plt.show()` in some contexts (though not needed in Jupyter inline mode). Plotting very large data without downsampling (can be slow). Overlapping plots when you forget to start a new figure or subplot (use `plt.figure()` or `plt.clf()` to clear). If using the OO interface, forgetting to apply methods on the correct Axes (`ax.plot()` vs `plt.plot()` when mixing styles). Also, not all plotting functions automatically create a figure, so ensure one exists (calling `plt.plot` will implicitly create one).

```
import numpy as np
import matplotlib.pyplot as plt

# Line Chart – Steps and Calories over Months
months = np.arange(1, 13)
steps = np.array([220000, 200000, 250000, 270000, 300000, 310000, 290000,
280000, 260000, 240000, 230000, 250000])
calories = np.array([68000, 64000, 72000, 76000, 82000, 85000, 80000, 78000,
74000, 70000, 69000, 73000])
plt.figure(figsize=(6,4))
plt.plot(months, steps, label='Steps', marker='o')
plt.plot(months, calories, label='Calories', marker='s')
# Highlight the month with highest steps
max_idx = np.argmax(steps)
plt.plot(months[max_idx], steps[max_idx], 'ro')  # red dot at peak steps
plt.title("Monthly Steps and Calories")
plt.xlabel("Month"); plt.ylabel("Count")
plt.axhline(0, color='gray', linewidth=0.8)      # baseline at 0 for
reference
plt.legend()
plt.show()

# Bar Chart – Average Sleep Hours per Month
sleep_hours = np.array([7.1, 6.9, 7.3, 7.0, 7.2, 7.4, 6.8, 6.9, 7.1, 6.7,
7.0, 7.2])
plt.figure(figsize=(6,4))
plt.bar(months, sleep_hours, color='skyblue')
plt.axhline(7, color='red', linestyle='--', label='Recommended 7h')
plt.title("Average Sleep Hours (Monthly)")
plt.xlabel("Month"); plt.ylabel("Hours")
plt.legend()
plt.show()
```

In the line chart above, we plotted two lines (Steps and Calories) against Month, and highlighted the peak activity month with a red marker. In the bar chart, we plotted average sleep hours for each month and added a horizontal dashed line at 7 hours to indicate the recommended threshold for healthy sleep.

```
# Scatter Plot – Heart Rate vs Sleep, colored by Steps
heart_rate = np.array([75, 78, 74, 76, 73, 72, 77, 76, 75, 79, 78, 74])  #
average heart rate each month
plt.figure(figsize=(5,4))
plt.scatter(sleep_hours, heart_rate, c=steps, cmap='viridis')
plt.title("Avg Heart Rate vs Sleep Hours")
plt.xlabel("Sleep Hours"); plt.ylabel("Avg Heart Rate")
plt.colorbar(label="Steps")  # color scale indicating Steps
plt.show()

# Pie Chart – Yearly Steps vs Calories Contribution
total_steps = steps.sum()
total_cals = calories.sum()
plt.figure(figsize=(4,4))
plt.pie([total_steps, total_cals], labels=['Steps', 'Calories'],
autopct='%1.1f%%',
        colors=['#66b3ff', '#ff9999'])
plt.title("Yearly Steps vs Calories")
plt.show()
```

The scatter plot uses color (`c=steps`) to encode a third dimension (monthly Steps) in the 2D plot of heart rate vs sleep hours – months with more steps appear in a different color. The pie chart compares total steps vs total calories burned in the year, with percentage labels (`autopct='%1.1f%%'` formats the percentages).

```
# Subplots (2x2 grid combining charts)
fig, axs = plt.subplots(2, 2, figsize=(8, 8))
# (0,0) Line plot for Steps and Calories
axs[0,0].plot(months, steps, label='Steps')
axs[0,0].plot(months, calories, label='Calories')
axs[0,0].set_title("Steps & Calories"); axs[0,0].legend()
# (0,1) Bar plot for Sleep Hours
axs[0,1].bar(months, sleep_hours, color='skyblue')
axs[0,1].axhline(7, color='red', linestyle='--')
axs[0,1].set_title("Sleep Hours")
# (1,0) Scatter plot (Heart Rate vs Sleep, colored by Steps)
sc = axs[1,0].scatter(sleep_hours, heart_rate, c=steps, cmap='viridis')
axs[1,0].set_title("Heart Rate vs Sleep")
axs[1,0].set_xlabel("Sleep Hours"); axs[1,0].set_ylabel("Heart Rate")
fig.colorbar(sc, ax=axs[1,0], label='Steps')  # colorbar for scatter
# (1,1) Pie chart for Steps vs Calories
axs[1,1].pie([total_steps, total_cals], labels=['Steps','Calories'],
             autopct='%1.1f%%', colors=['#66b3ff','#ff9999'])
axs[1,1].set_title("Steps vs Calories")
# Overall figure title and layout
fig.suptitle("Health & Fitness Analysis (2024)")
plt.tight_layout()
```

```
plt.savefig("health_report.png")
plt.show()
```

In the above code, we combined four plots into one figure with a 2x2 grid of subplots. We add a main title for the figure and adjust the layout. Finally, we save the figure to a PNG file. This approach is useful for creating dashboards or summary charts in reports.

Seaborn

Core Concepts: Seaborn is a high-level visualization library built on Matplotlib, designed to make it easier to create attractive and informative statistical graphics. It integrates well with Pandas DataFrames, allowing you to specify data columns by name. Seaborn comes with default themes that make plots look more polished out of the box, and provides specialized plot types for statistical analysis (distribution plots, regression plots, categorical plots, etc.). Typically, it's imported as `import seaborn as sns`. You might also call `sns.set_theme()` (or `sns.set_style('whitegrid')`) to set a theme.

Plot Types in Seaborn: - **Scatter and Line:** `sns.scatterplot(x=..., y=..., data=df, hue=..., style=...)` plots points, with optional color grouping by `hue` category and different markers by `style`. `sns.lineplot` works similarly for lines (and can aggregate data by category). - **Categorical Plots:** Seaborn has several: `sns.barplot(x=cat, y=val, data=df)` shows mean (or other estimator) of `val` for each category `cat` (with error bar by default); `sns.boxplot(x=cat, y=val)` shows distribution per category as a box-and-whisker; `sns.violinplot` shows a violin (smoothed distribution) per category; `sns.stripplot` or `swarmplot` show individual data points. - **Distribution Plots:** `sns.histplot(data, bins=...)` for histogram (with optional `kde=True` for overlayed density curve), `sns.kdeplot` for just the density, `sns.distplot` (deprecated in latest versions). - **Matrix Plots:** `sns.heatmap(data)` to show a color-encoded matrix (often used for correlation matrices; use `annot=True` to label cell values [4]). - **Pair Plot:** `sns.pairplot(df, hue='category')` automatically plots all numeric pairwise relationships in the DataFrame in a grid of scatterplots [5] [6], with diagonal showing distributions. The `hue` parameter will color points by the given categorical variable (e.g., different classes) [6]. - **Joint Plot:** `sns.jointplot(x=..., y=..., data=df, kind='scatter' or 'kde', hue=...)` shows the joint distribution of two variables (scatter or density) along with their marginal distributions (histograms or density curves on the sides). - **PairGrid / FacetGrid:** More advanced grids for plotting multiple plots conditioned on subsets of data (e.g., plotting a grid of plots for combinations of two categorical variables). - **Regression Plot:** `sns.regplot(x, y, data=df)` plots a scatter and fits a regression line (with confidence interval) – useful for quick insight into trend.

Using Hue and Other Semantic Mappings: A key feature in Seaborn is the ability to add semantics like color (`hue`), marker style (`style`), or size (`size`) to represent additional dimensions. For example, in a scatterplot, `hue='target'` might color points by whether a patient has heart disease or not. This helps distinguish groups in visualizations [6]. Seaborn automatically provides a legend for these. In many plots, `hue` will create side-by-side comparisons (e.g., `sns.boxplot(x='sex', y='chol', hue='target', data=df)` would draw two boxes per sex category, one for each target group).

Customization and Themes: Seaborn allows global styling with `sns.set_theme()` or specific context with `sns.set_context('talk')` (for bigger labels suitable for presentations, etc.). You can control color palettes via `palette` parameter or use built-ins like `'viridis', 'coolwarm',`

`'pastel', 'deep'` etc. Many Seaborn plots are actually wrappers around Matplotlib axes, so you can often tweak further with Matplotlib functions (e.g., `plt.xticks` or `ax.set_title` on the underlying Axes).

Common Patterns from Lab: For example, using the **heart disease dataset**: - Pairplot on a subset of features with `hue='target'` to see clustering of patients with/without disease. - Boxplots of blood pressure or cholesterol split by groups (to compare distributions). - Violin plots to visualize distribution shapes of a metric split by category (e.g., max heart rate by disease, or age by sex). - Correlation heatmap to identify which factors correlate with the target (heart disease presence). - Jointplot of two key variables (e.g., age vs max heart rate) to see their relationship and distribution of each, possibly colored by target. - Barplots to compare group means (e.g., chest pain type vs average max heart rate, broken down by disease status).

Tips: - Many Seaborn functions return a Matplotlib Axes (or FacetGrid); you can capture it to further adjust if needed: `ax = sns.boxplot(...)`. - Use `sns.pairplot` for quick EDA (exploratory data analysis) of all pairs of variables – it's a one-liner that can replace many individual plots. - Use `sns.heatmap(corr, annot=True, fmt=".2f")` for a correlation matrix to see numeric values in each cell [4]. - **Statistical Estimators:** Some plots like `sns.barplot` or `sns.pointplot` compute an estimator (mean by default) and confidence interval. You can change the estimator via the `estimator` parameter (e.g., `estimator=np.median` to plot medians). - For **violinplot** or **boxplot** with two-category comparisons, you can use `hue` or the `split=True` parameter on violinplot to split the violin into two halves by hue category for easier comparison. - If the dataset is large, pairplot and jointplot can be slow – consider sampling or using `sns.histplot` with 2D binning (`sns.histplot(x, y, data, bins=30)`).

Common Pitfalls: Not specifying `data=df` and column names in Seaborn functions (Seaborn can take either long-form DataFrame or NumPy arrays; using DataFrame with column names is more convenient). Overplotting in scatterplots for large data – consider using transparency `alpha` or smaller `s` (point size). Forgetting that some Seaborn functions (like pairplot or jointplot) create their own Figure, so if you call them one after another, they will make multiple figures (you might need to adjust with `plt.close()` if in a script). Also, if using Jupyter, remember to `%matplotlib inline` or `%matplotlib notebook` to see plots inline.

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
# Sample small heart disease dataset (subset of rows for demo)
heart_df = pd.DataFrame({
    'age':    [63, 41, 67, 62],
    'sex':    [1,  0,  1,  0],   # 1=male, 0=female
    'cp':     [3,  1,  0,  0],   # chest pain type (0-3)
    'trestbps':[145,130,160,140],# resting blood pressure
    'chol':   [233,204,286,268], # serum cholesterol
    'thalach':[150,172,108,160], # max heart rate achieved
    'target': [1,  1,  0,  0]    # 1=disease, 0=no disease
})
# 1. Pairplot for selected features, colored by target
sns.pairplot(heart_df[['age','chol','thalach','target']], hue='target')

# 2. Boxplots: Compare distributions by category
```

```
plt.figure(figsize=(6,4))
sns.boxplot(x='target', y='trestbps', data=heart_df)
plt.title("Resting BP by Heart Disease (target 0/1)")
plt.figure(figsize=(6,4))
sns.boxplot(x='sex', y='chol', data=heart_df)
plt.title("Cholesterol by Sex")
plt.show()

# 3. Violin plots: Show distribution shape
plt.figure(figsize=(6,4))
sns.violinplot(x='target', y='thalach', data=heart_df, split=True)
plt.title("Max Heart Rate by Heart Disease")
plt.figure(figsize=(6,4))
sns.violinplot(x='sex', y='age', data=heart_df, split=True)
plt.title("Age Distribution by Sex")
plt.show()

# 4. Correlation Heatmap
corr =
heart_df[['age','sex','cp','trestbps','chol','thalach','target']].corr()
sns.heatmap(corr, annot=True, fmt=".2f")
plt.title("Correlation Matrix (Heart Data)")
plt.show()

# 5. Jointplot: Age vs Max Heart Rate, colored by disease
sns.jointplot(x='age', y='thalach', data=heart_df, hue='target',
kind='scatter')
sns.jointplot(x='age', y='thalach', data=heart_df, hue='target', kind='kde')
plt.show()

# 6. Barplots: Compare means across categories with hue
plt.figure(figsize=(6,4))
sns.barplot(x='cp', y='thalach', hue='target', data=heart_df,
estimator=np.mean)
plt.title("Chest Pain Type vs Avg Max HeartRate")
plt.figure(figsize=(6,4))
sns.barplot(x='sex', y='chol', hue='target', data=heart_df,
estimator=np.mean)
plt.title("Sex vs Avg Cholesterol")
plt.show()
```

Let's break down what the Seaborn code does:

1. **Pairplot:** We use `sns.pairplot` on a subset of the heart dataset (age, chol, thalach, target). With `hue='target'`, points are colored by the target class (heart disease or not), making any separation patterns visible at a glance. This single call creates a grid of scatter plots for each pair of features and histograms on the diagonal.

2. **Boxplots:** The first boxplot compares resting blood pressure (`trestbps`) distributions between patients with and without heart disease (`target` 1 vs 0). The second compares cholesterol

levels by sex (male vs female). We see medians, IQRs, and potential outliers for each group. Boxplots are great for spotting differences in medians or variability across categories.

3. **Violin Plots:** Similar to boxplots but show the full distribution shape (kernel density estimate) for each category. We used `split=True` to split the violin when there is a hue (here we directly used `x` as the category of interest since our data is small; alternatively, we could have used hue if we had a second category). The first violin plot shows the distribution of max heart rate for patients with vs without heart disease. The second shows the age distribution by sex. The violin's width indicates density of observations at that value.

4. **Heatmap:** We compute a correlation matrix for the features and then plot it with `sns.heatmap`. We set `annot=True` to show correlation coefficients in each cell, formatted to 2 decimal places (`fmt=".2f"`). This helps identify which variables are correlated with each other and with the target. For example, we might see that `thalach` (max heart rate) is negatively correlated with `age` (older patients tend to achieve lower max heart rates), or that having heart disease (`target`) might correlate with certain features (in a larger dataset, we might find, say, higher cholesterol correlates with disease presence).

5. **Jointplot:** We create two joint plots for age vs. max heart rate. The first (`kind='scatter'`) shows a scatterplot of these variables with points colored by disease status. The second (`kind='kde'`) overlays the kernel density estimate showing the concentration of points. These plots reveal relationships (e.g., perhaps younger patients reach higher heart rates, and we might observe clustering of disease vs no-disease in this space). Jointplot automatically includes marginal distributions (histograms or density plots on the top and right axes for the single variables).

6. **Barplots:** Here we compare averages across categories with hue. The first barplot shows average max heart rate for each chest pain type (`cp` 0,1,2,3), separated by whether the patient has heart disease or not (hue by target). The second barplot shows average cholesterol by sex, again split by disease status. Barplots use `estimator=np.mean` by default (we explicitly set it for clarity), and would include error bars (confidence intervals) if the data had multiple points per category in this small example (with more data, you'd see error bars). This kind of plot helps to compare group means directly.

Note: In practice, you'd typically use the full dataset for these visualizations. Here we used a small sample for demonstration, but with more data, patterns become clearer. Always label axes and add titles or legends as needed for clarity (Seaborn often adds legends automatically when using hue). By combining these tools, you can quickly perform exploratory data analysis: pairplots for broad overview, heatmaps for correlations, and specific plots (box/violin/bar) to drill into relationships between variables and the target.

---

[1] [2] Python Dataclasses - Python Cheatsheet
https://www.pythoncheatsheet.org/cheatsheet/dataclasses

[3] pandas.DataFrame.pipe — pandas 2.3.2 documentation
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pipe.html

[4] seaborn.heatmap — seaborn 0.13.2 documentation
https://seaborn.pydata.org/generated/seaborn.heatmap.html

5 6 seaborn.pairplot — seaborn 0.13.2 documentation
https://seaborn.pydata.org/generated/seaborn.pairplot.html